Bachelorarbeit

im Studiengang Statistik

an der Ludwig-Maximilians-Universität München

Department Institut für Statistik

Visualization and replay memory database design for reinforcement learning in R

Written by
Sebastian Gruber

Duty date
24th July 2018

Supervision
Prof. Dr. Bernd Bischl
Xudong Sun

Abstract

In this thesis, a basic intro to reinforcement learning, deep learning and database systems will be given. Reinforcement learning describes methods learning on data generated from acting in an environment. Unless conventional machine learning algorithms, these kind of artificial learners are meant to start with zero data. Additionally to the theoretical parts in this thesis, there will be several implementations in R not only for demonstration purposes, but also to help storing the generated data plus understanding the learning process in retrospective.

Contents

1 Introduction

Due to the increase of computing power and the introduction of new powerful algorithms in the deep learning subdomain, machine learning as a whole has experienced a substantial uplift of interest - not only in the research world, but also on the business side of things in form of millions of dollars of investment [17]. This broad hype has led many key figures in society to believe that humanity is stepping into a new age of data-driven artificial intelligence, although it is questionable to call algorithms trying to fit more or less random patterns in a limited pile of numbers as intelligence. Especially, because a key component of intelligence - generalisation - will only be to some extend achievable with the engineering and supervision of highly skilled craftmanship (the so called *Data Scientists*) if - and only if - the available data does allow general patterns to be find. There is also the additional and highly time-inefficient challenge of cleaning at first glance useless data to something a machine learning algorithm can even run on because universal standards for storing data are lacking and a general carelessness for the data quality during the data generation process is still common.

And this is the part where Reinforcement Learning comes to shine. As a subcategory of machine learning algorithms it has its similarities with other machine learning methods and even puts some of these into highly use, but with one single big difference allowing to cirumvent some of the above mentioned disadvantages: It starts with zero data.

Unless conventional algorithms like in supervised or unsupervised learning, these kind of artificial learners start in a virtual environment and generate their own data by acting in there and experiencing the consequences of their actions in terms of change of this environment plus a reward upon reaching a predefined goal. Not only does that allow to learn a solution of problems being too high dimensional and dynamic for exhaustive search methods (like the chinese game *Go*), but this also means the algorithm tries to learn the optimal behaviour based on a whole environment and not on scarse, badly balanced or biased data points, achieving better generalisation as a consequence. Thanks to this, quicker and better learned solutions than through the usage of other algorithms based on natural/human data become possible.[14]

Of course reinforcement learning has its own kind of problems. It is absolutely mandatory that every task exists as a (time consumingly) crafted virtual simulation as a bag for pulling the data out, plus this data may require very high amount of storage, plus the learning process may be very unstable (the algorithm can even fail to converge to a solution), plus even if a good solution is reached, it's extremely difficult to reason about that. These difficulties are a major reason why reinforcement learning is still mostly stuck in academic research, while being wholeheartedly ignored by most of the real world without access to a specialized research department. This bachelor thesis will not be about how to program a virtual environment in a quick manner, but I will try to tackle the other above mentioned issues by providing a storage solution and a visualization app to review the learning process.

First of all, there will be quick introduction to the theoretical backgrounds of reinforcement learning plus an in R implemented basic example, giving a better understanding than the shallow words used so far. This follows up with an explanation of some basic knowledge around artifical neural networks, that are heavily used be reinforcement learning algorithms, and further theory about relational databases to understand how the increase of storage efficiency in the implementation part is achieved for the replay memory (synonym for the stored generated data pile). After that, another chapter describes the visualization possibilities of the learning process introduced by a shiny app. This implementation will be done as addition for the reinforcement learning specific R package *rlR* (authored by Xudong Sun [15]).

2 Preliminaries

2.1 Introduction to reinforcement learning

Before any task can be handled by a reinforcement learning algorithm, it is absolutely required for the task to be specified as a Markov Decision Process (MDP). To define a MDP a finite set $\mathcal{S}$ of states $s \in \mathcal{S}$ is needed, in where it is possible to transition stepwise between state i to state j by a given probability $\mathcal{P}_a(i,j)$ (called transition probability) with a stochastic reward $r \in \mathcal{R} \subset \mathbb{R}$ as additional result - although future rewards suffer from a discount factor $\gamma \in [0,1]$. Furthermore these transition probabilities are influenced for each state by taking actions $a \in \mathcal{A}$ - $\mathcal{A}$ also being a finite set. To sum this up, a MDP is specified as a five-tupel of $(\mathcal{S}, \mathcal{A}, \mathcal{P}_a, \mathcal{R}, \gamma)$. [16]
It is important to note that no matter what transition happens, the system can never be left. This means that given a certain state, the cumulative probability to move to a new state is always equal 1. Note that it's also possible to transition to the same state again - leading to no state change at all. This special case is also treated as a transition to a new state. Of course it is also possible for single transitions to have probability of zero. This simply means a specific state can not be reached from this state. If all transitions are zero in probability except the transition to the state itself, the state is called a terminal state because change is impossible.
To make this compliant with real world tasks, we simply define every possible view-/datapoint during a task as a state and the whole environment around the task as the system. For example if we have a robot in a room, we can either define each image frame captured by a camera as a state or - if the robot always exists on a grid with discrete values - the positional coordinates after every move of itself as states. In both scenarios it is important to summarize all the knowledge (the image or the coordinates) into a single state.
Additionally, since we want an artificial and smart actor in an environment and not just simply transition randomly from state to state without goal or reasoning, we have to take the predefined numerical reward from each transition into account and the possible actions we can take as condition of how the next state plus reward may look like - we don't look at transitions from state to state alone, but rather from states paired with actions to states paired with rewards. This gives us the flexibility on the one hand to say that actions in the same state have consequences on what next state will be reached while on the other hand to assume different kind of rewards with different probabilities for the next state.
The following definition is a mathematical way of expressing the just explained requirements for the *transition probabilities* [16, p. 38]:

$$p(s', r|s, a) \coloneqq \mathbb{P}(S_t = s',\, R_t = r \,|\, S_{t-1} = s,\, A_{t-1} = a) \tag{1}$$

in which s', $s \in \mathcal{S}$, $r \in \mathcal{R}$ and $a \in \mathcal{A}(s)$
with sets of states, actions and rewards ($\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$) and t as the transition iteration. This term basically tells us all about what is important for a Reinforcement

Learning algorithm to work - an existing/current state, an action to do, a next state to transition to (can be the same state) plus a reward after the transition happened. As described before, it is essential that the following holds true:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \text{ for all } s \in \mathcal{S},\ a \in \mathcal{A}(s) \tag{2}$$

Otherwise, the agent (synonym for acting algorithm) could possibly leave the environment. With the help of 1 it is possible to calculate all other desired attributes in the MDP. Often it is useful to simply calculate the state-transition probabilities [16, p.38]:

$$p(s'|s, a) \coloneqq \mathbb{P}(S_t = s' \,|\, S_{t-1} = s,\ A_{t-1} = a) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \tag{3}$$

Or - another important attribute - the expected reward given the agent is in a certain state and applies a certain action:

$$r(s, a) \coloneqq \mathbb{E}(R_t \,|\, S_{t-1} = s,\ A_{t-1} = a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a) \tag{4}$$

Besides that we can also calculate the expected reward once a certain state is reached from a given previous state and action:

$$r(s, a, s') \coloneqq \mathbb{E}(R_t \,|\, S_{t-1} = s,\ A_{t-1} = a,\ S_t = s') = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)} \tag{5}$$

But since we want to solve a whole environment the algorithm needs something to optimize for, so we won't get far by only looking at single state transitions. To make the agent strive towards a single goal it is important to gather the rewards of a chain of transitions together. This is called the Return G_t of the current iteration t and it's nothing else except the sum of all rewards from future transitions multiplied by a constant γ [16, p. 43]:

$$G_t \coloneqq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{6}$$

$0 \leq \gamma \leq 1$, $T \leq \infty$ (in which either $\gamma = 1$ or $T = \infty$) γ has two purposes here: On the one hand it tells the agent how short- or long sightened it should act, while on the other hand it makes sure the expected return being calculable by keeping it smaller than infinite. The Return can also be written recursively:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{7}$$

Now, after specifying what goal the agent should follow, we want to set how it acts to achieve this goal. For this we specify a policy function π acting in the most

general case as a probability density for the actions a the agent should commit to in a certain state s:

$$\pi(a|\, s) \;= P(\mathcal{A}_t = a|\, \mathcal{S}_t = s)$$

The definition of the policy function above is the case of a stochastic policy and the more general form of a deterministic policy. A deterministic policy is just a strict mapping from a state to an action:

$$\pi(s) = a, \text{ with } s \in \mathcal{S} \text{ and } a \in \mathcal{A}$$

The latter basically makes the agent always perform the same action in the same state. This is highly desired if the agent should maximize its overall reward, but it has also a lack of exploration as consequence. Since the agent is supposed to gather more information about an environment and basically never starts with full knowledge, stochastic policies like $\epsilon - greedy$ have become common in Reinforcement Learning algorithms. Upon this we can now calculate the expected return of a certain state under the condition of the agent following policy π [16, p. 46]:

$$v_\pi(s) \;\coloneqq\; \mathbb{E}_\pi(G_t \,|\, S_t = s) \;=\; \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right) \tag{8}$$

for all $s \in \mathcal{S}$.

This is called *state value function* and allows us to quantify the expected return an agent with a set policy recieves in the future for every state. Furthermore it is now possible to differentiate between states in terms of how desired each one is for achieving maximum return. Since the goal for an agent is to find the optimal policy to gather the highest return possible, we can see finding the real expected returns for each state as an important milestone for trivializing our main task.
Additionally, to make things even more straight forward for an artifical actor, we can calculate the expected return of specific actions done in specific states with the same condition of the agent following policy π again:

$$q_\pi(s, a) \;\coloneqq\; \mathbb{E}_\pi(G_t \,|\, S_t = s,\, A_t = a) \;=\; \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s,\, A_t = a\right) \tag{9}$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$. The above definition is called *action value function* and holds at least as much importance as (8) - while (8) only gives us information about how desirable a state is for maximizing reward, (9) tells us how desirable it is to make a certain action in a certain state, giving us a direct chain of orders (actions in context of a policy) to follow for transitioning from state to state while reaching optimal rewards.
Of course knowing the real values of (8) or (9) would make any task trivial, but (un)fortunately this is basically never the case in the real world. These functions

always have to be explored iteratively by an agent to find out their true appearance and therefore generate more precise (and hopefully more rewarding) solutions for the policy. With the use of the transition probabilities it is also possible (with the help of (7)) to write both state value and action value functions recursively based on their own values of the next state:

$$\begin{aligned} v_\pi(s) &\coloneqq \mathbb{E}_\pi(G_t \,|\, S_t = s) && (10)\\ &= \mathbb{E}_\pi(R_{t+1} + \gamma G_{t+1} \,|\, S_t = s) && (11)\\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\big(r + \gamma \mathbb{E}_\pi(G_{t+1} \,|\, S_{t+1} = s')\big) && (12)\\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\big(r + \gamma v_\pi(s')\big) && (13) \end{aligned}$$

for all $s \in \mathcal{S}$.
The last two equations are two different forms of the so called *bellman equation* [16, p. 47]. Analogous, it also exists for the action value function:

$$\begin{aligned} q_\pi(s, a) &\coloneqq \mathbb{E}_\pi(G_t \,|\, S_t = s,\, A_t = a) && (14)\\ &= \mathbb{E}_\pi(R_{t+1} + \gamma G_{t+1} \,|\, S_t = s,\, A_t = a) && (15)\\ &= \sum_{s'} \sum_r p(s', r|s, a)\big(r + \gamma \max_{a'} \mathbb{E}_\pi(G_{t+1} \,|\, S_{t+1} = s',\, A_{t+1} = a')\big) && (16)\\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\big(r + \gamma \max_{a'} q_\pi(s', a')\big) && (17) \end{aligned}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.
Furthermore to get the most reward out of an environment by following a specific policy, the agent has to find the *optimal policy* π_* out of all possible policies π. It is defined by [16, p. 50]:

$$\pi_*(s) \coloneqq argmax_\pi v_\pi(s), \text{ for all } s \in \mathcal{S} \tag{18}$$

This can be interpreted as that there doesn't existing any other policy having a higher state value in any existing state than the optimal policy. Although, it is not required for the optimal policy to be unique. Based on (18) and (7) the *optimal state-value function* can be defined by the following equation:

$$v_*(s) \coloneqq \max_\pi v_\pi(s), \text{ for all } s \in \mathcal{S} \tag{19}$$

And analogous, the same follows for the *optimal action-value function*:

$$q_*(s, a) \coloneqq \max_\pi q_\pi(s, a), \text{ for all } s \in \mathcal{S},\, a \in \mathcal{A}(s) \tag{20}$$

Again, the optimal action-value function can be expressed in terms of the expectation of the reward plus optimal state-value function of the next state:

$$q_*(s,a) \;=\; \mathbb{E}(R_{t+1} + \gamma v_*(S_{t+1}) \,|\, S_t = s,\, A_t = a) \tag{21}$$

With the help of the transition probabilities we can now define the *Bellman optimality equation* for v_* by using the above formulas and (17):

$$\begin{aligned} v_*(s) \;&=\; \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) && (22)\\ &=\; \max_a \mathbb{E}_{\pi_*}(G_t \,|\, S_t = s,\, A_t = a) && (23)\\ &=\; \max_a \mathbb{E}_{\pi_*}(R_{t+1} + \gamma G_{t+1} \,|\, S_t = s,\, A_t = a) && (24)\\ &=\; \max_a \mathbb{E}(R_{t+1} + \gamma v_*(S_{t+1}) \,|\, S_t = s,\, A_t = a) && (25)\\ &=\; \max_a \sum_{s',r} p(s', r|s, a)\big(r + \gamma v_*(s')\big) && (26) \end{aligned}$$

This equation has the benefit of not requiring a policy to calculate v_* as it expresses the fact of the optimal policy simply being equal to the best action in terms of highest expected return.
Once more, the bellman optimality equation also exists for q_*:

$$\begin{aligned} q_*(s,a) \;&=\; \mathbb{E}(R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \,|\, S_t = s,\, A_t = a) && (27)\\ &=\; \sum_{s',r} p(s', r|s, a)\big(r + \gamma \max_{a'} q_*(s', a')\big) && (28) \end{aligned}$$

2.2 Implementation of basic reinforcement learning alogrithm

So, what we know from the last chapter is if an agent wants to find an optimal policy in an environment either the unknown state values or the action values do have to be as close as possible to their real values. In this chapter a small algorithm that focuses on finding out the nature of the action value function by running iteratively on a small discrete environment is being introduced. The idea is to store the more or less correctly guessed action values Q inside a tabular (only works for small state spaces) and for each new experience of a state-action pair to update the Q value inside the tabular with the just encountered reward [16, p. 133]. To make words more discrete, the above happens in form of the following pseudo code line after every step the agent took following a policy with $(S,\, A,\, R,\, S') \in \mathcal{S} \times \mathcal{A} \times \mathcal{R} \times \mathcal{S}$:

$$Q(S,\, A) \;=\; Q(S,\, A) + \alpha\,[R + \gamma\; max_{a \in \mathcal{A}}\, Q(S',\, a) - Q(S,\, A)] \tag{29}$$

The policy is kept to be ϵ-greedy, meaning the agent will either act greedy by probability of $1 - \epsilon$ or randomly otherwise. This ensures the states near the agents current

optimal behaviour are also explored to some degree - because the momentary policy may be pretty bad overall due to a lack of knowledge of the true action values. By doing actions and thus moving through the environment plus updating its knowledge (the Q tabular), the agent slowly learns the correct values of the true action value function and can find a best solution for the environment. Of course with this being a very simply algorithm, the statement only holds true for also simple environments. To give insight of how this looks in action, the algorithm is implemented and executed in the programming language R. The following code snippet gives a minimal example of how the core functionality of a single episode will look like written in a executable program. Mind that the agent stops his episode either if a terminal state is reached or if 100 steps were done without achieving anything.

```
for (iteration in 1:100) {
  rand_a = sample(1:4, 1)
  best_a = sample(which.max(Q[state, ]), 1)
  a = sample( c(best_a, rand_a), 1, prob = c(1-eps, eps) )
  step = env$step(a - 1)
  s_1  = step[[1]] + 1    # new state
  r    = step[[2]]        # reward
  done = step[[3]]
  Q[s,a] = Q[s,a] + alpha * (r + gamma * max(Q[s_1,]) - Q[s,
     a])
  s    = s_1              # update old state
  if (done) break
}
```

—End of code—

The environment has a simple chessboard alike layout with 4×12 squares and was introduced in the gym package [2]. The start state is the left lower corner, the target state the right lower one - the latter also holds a positive reward if stepped onto it. Between the start and the target state is a cliff, giving the agent a big minus reward if he steps into it and moving him back to the start state. All other states are free to move around. As a consequence of this definition, to get the most reward out of this environment one has to find the shortest route from the start to the target state. A visualization of this environment is given in the following:

o	o	o	o	o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o	o	o	o	o
x	C	C	C	C	C	C	C	C	C	C	T

From here on several visualizations of the decisions in the different states based on a greedy policy will be depicted after a specific amount of learning episodes. Note

there are simply so many up-arrows in the beginning because at first all decisions have the same action values, so the first action (up) is simply chosen. The decisions the agent makes after one episode:

↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
→	↑	↑	↑	↑	↑	↑	↑	↑	↑	←	→
→	↑	↑	↑	↑	↑	↑	↑	↑	↑	→	↓
→	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑

For some few states the optimal decision is already found, but most are still wrong which is no surprise after only one episode.
The decisions after 150 episodes:

↓	→	↓	↓	↓	↓	→	↓	↓	↓	↓	↓
←	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
→	→	→	→	→	→	→	→	→	→	→	↓
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑

Above, one can see that the agent found out the perfect route from the start to the target location. There are still other states being off, but these only matter from here on if the agent moves onto them by random.
And the updated decisions after 500 episodes:

→	→	→	→	→	→	→	→	→	→	→	↓
→	→	→	→	→	→	→	→	→	→	→	↓
→	→	→	→	→	→	→	→	→	→	→	↓
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑

At 500 episodes based on the tabular above it is clear the agent found the optimal policy for every state existing in the environment. No matter at what position he is, the right choice will always be made.
Furthermore the return after every episode is plotted as a time series graph in figure 1. Highly noteworthy is the change of the slope around episode 150 - this is exactly the spot when the agent discovered the best greedy solution. The further minor improvements can be explained by the agent ending up in a state off the perfect route due to random choices.

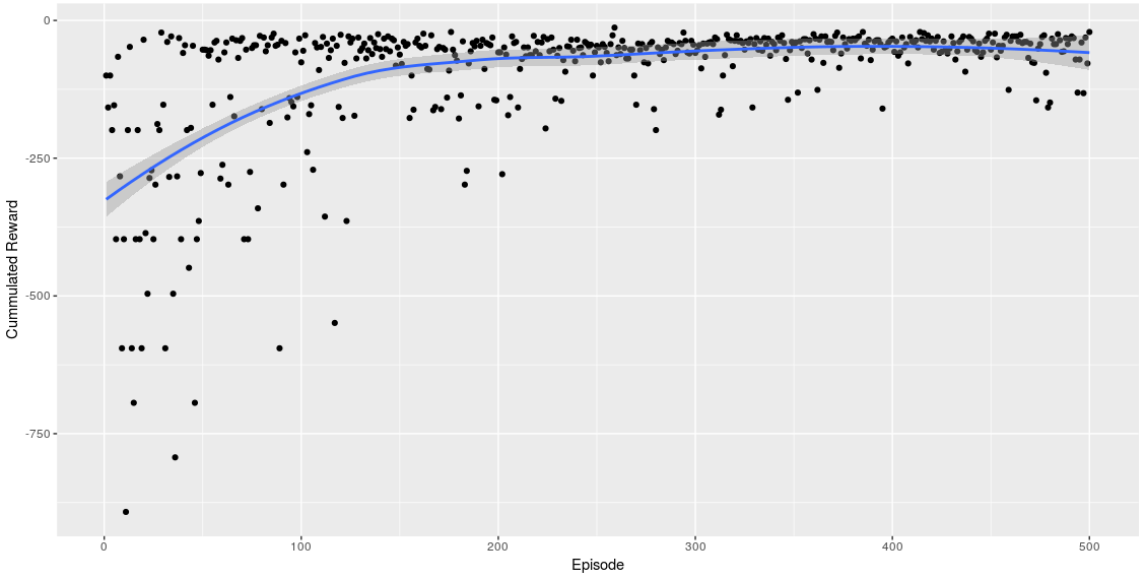

Figure 1: Change of the cummulated rewards over training episodes

2.3 Introduction to artificial neural networks

Most problems comming from the real world have a quasi-infinite amount of states making tabular learning impossible to implement without further manual discretization and thus a loss of information of the environment. To circumvent this problem the values of action/state functions aren't stored in a tabular anymore, but rather expressed as a function approximating the real value function. This is done by using a machine learning algorithm iteratively adapting its weight to reduce the error occuring by wrong estimations. The most common choice for a task like this are artificial neural networks due to their high flexibility and scalability for tasks of high complexity like making predictions on image data.

Artificial Neural Networks exist as a link of several single nodes (analogous to neurons in nature). The name is inspired by brain research, but in machine learning it's nothing more than a weight vector w multiplied on a numeric input vector x with a bias weight b and an activation function f (like for example ReLU(.) $= \max(0, .)$) on top [8, p. 167]:

$$y = f(\, w^T x \, + \, b\,) \tag{30}$$

$$w,\, x \in \mathcal{R}^n$$

$$b \in \mathcal{R}$$

In Figure 2 the calculations of a single node are presented in a very graphic way to make the connection to brain science even more clearer - though this kind of depiction is unhandy for bigger networks and that's why the scheme of Figure 3 is used from now on. The weights of w and b are initialized randomly and are iteratively changed during a training phase to approximate the structural link between a target

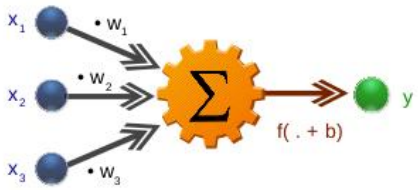

Figure 2: Graphical illustration of the calculations in a single node

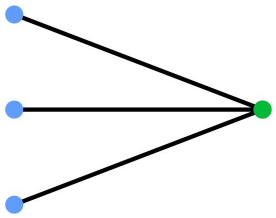

Figure 3: Minimalistic depiction

variable y and its depending data x - a more extensive section on the training will follow further down.

2.3.1 Layer types

To recieve a high dimensional output vector y instead, the weight vector w is simply expanded to a weight matrix W and the bias scalar to a bias vector b with each row in W and each entry of b corresponding exclusively to an output dimension. We then call the function $f(x;\, W,\, b)$ a layer.
Formulating this in pseudo code gives [8, p. 191]:

$$y = f(x;\, W,\, b) = f(\, Wx \,+\, b\,) \tag{31}$$
$$W \in \mathcal{R}^{mxn}$$
$$b \in \mathcal{R}^{m}$$

For example if a three dimensional output vector is desired for an eight dimensional input vector, the W will have 24 entries - this specific case is visualized in figure 4.

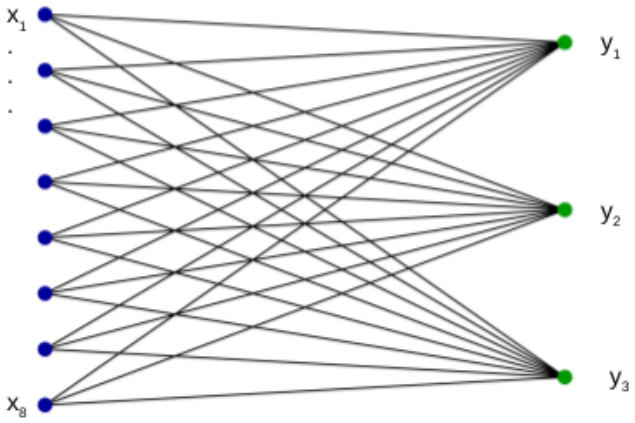

Figure 4: Scheme of a layer with three nodes

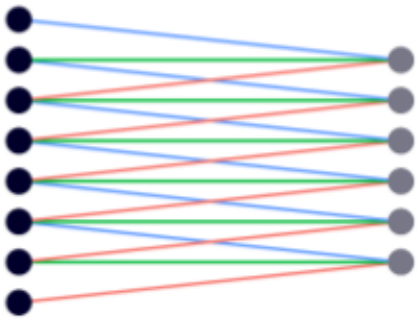

Figure 5: Convolutional layer with kernel size $h = 3$. Connections with the same colours share the same weights.

A weight matrix in which every entry is adaptable and adapted individually by the training process is called a *fully connected layer*.

But such a layer where every node-input-dimension pair has its own weight also has a big disadvantage, namely the lack of scalability for higher dimensional input like image data. Images usually grow roughly quadratic in their pixel amount (because for a sharper image one has to increase the resolution not only in the x-, but also in the y-direction), as a result the amount of weights required to simply input a vector representing the pixels of an image also grows quadratic. So, to make neural networks also work on such cases, there was another layer type introduced by the scientific community - called *convolutional layer* [8, p. 247]. This kind of layer requires far less individual weights, because by design it only has individual weights inside a ẃindow(́called *kernel* from here on) of predefined size h [8, p. 325]. This kernel then moves over all possible neighborhoods of the input and creates new kinds of features (the output here is called f́eature map)́ as output for the next layer [9]. As a result this gives a reduction of dimensionality by a minimal amount of trainable weights - resulting in more stable and highly computational efficient training phases. Depending on the dimensionality of the input, this kernel may be one dimensional (in case of vector data) or two dimensional (in case of matrix data, like images). For introductional purposes only the one dimensional case will be explained from here on, as the two dimensional works the same, but is a lot harder to visualize. In figure 5 a convolutional layer is shown. The colours explain the weight dependencies, and as one can see only three weights have to be trained in this example even though the size of the input is way higher. [8, p. 326]

For the one dimensional case of a convolutional layer it is possible to explain the calculations once more as a weight matrix. The dimension of the latter will be $(n - h + 1) \times n$ - with n as the input vector length and h as the kernel size. In figure 6 and 7 are two examples of a fully connected and a convolutional weight matrix with same sized input vectors for comparison depicted. Again, this makes clear how much less trainable weights are required for a convolutional layer.

a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x

Figure 6: Weight matrix of a fully connected lyers with entries $a, ..., x \in \mathcal{R}$

a	b	c	0	0	0	0	0
0	a	b	c	0	0	0	0
0	0	a	b	c	0	0	0
0	0	0	a	b	c	0	0
0	0	0	0	a	b	c	0
0	0	0	0	0	a	b	c

Figure 7: Weight matrix of a convolutional layer with entries $a, b, c \in \mathcal{R}$

To furthermore reduce the size of the output vector (or feature map) an operation called max pooling takes place. This operation also uses a kernel for creating neighborhoods like the convolutional layer, but in this case non-overlapping ones. Then, simply the highest value in each neighborhood is taken, while all the others are discarded. The idea behind this is to only focus on important/relevant information by only giving the strongest signal in a neighborhood a pass-through to the next layer. [8, p. 330]

2.3.2 Network design

Now we know how a single layer looks like and since the input and output of such one are both vectors, we encounter the possibility of stacking several different layers on top of each other - meaning the output vector of one layer is (part of) the input vector of another one. A chain of layers like this is called an *Artificial Neural Network* (ANN). The last layer of an ANN is called the output layer and all layers beforehand are the so called hidden layers. [8, p. 163/164]
Overall an ANN can be seen as a function mapping an input (data) vector to an output space by an arbitrarily (but manually predefined) amount of adjustable weights θ:

$$y = ANN(x; \theta)$$

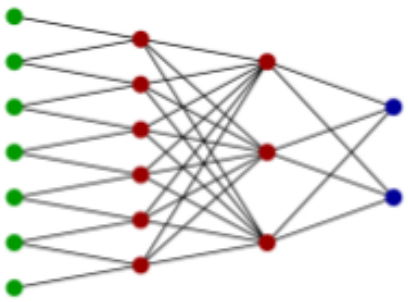

Figure 8: Example scheme of a convolutional neural network architecture

A more specific example of how a chain of layers is in figure 8 depicted. Here a 1D convolutional layer is put on the input vector, followed up by two fully connected layers plotting to a two dimensional output vector, like one is used for the fitting of binary classification tasks for example.
Writing this down as pseudo code ($*$ is matrix multiplication):

$$hidden\_layer_1 = f_1(W_1 * input\_vector + b_1)$$

$$hidden\_layer_2 = f_2(W_2 * hidden\_layer_1 + b_2)$$

$$output\_layer = f_3(W_3 * hidden\_layer_2 + b_3)$$

Here, *input_vector* would be our data vector x and *output_layer* the prediction $\hat{y}$. The forward processing of each values through the different layers is also called *forward propagation*. [8, p. 197]
Thanks to the recursive nature of such an architecture it is possible to create function fitters for prediction tasks with any kind of complexity. Configured and tuned the right way this can even lead to a predictive accuracy superior to human. But with great power comes great responsibility, meaning there are very important drawbacks of complex neural networks:

- In general ANNs have more weights then strictly required for a predictive task. This is because an ANN with perfectly adjusted weights may still have a bias and isn't able to fully fit the structure of data if not enough weights/layers are predefined. On the other hand, starting with too many weights only has an increased computational effort as disadvantage at first. As a consequence of this, ANNs are initiated with an highly generous amount of weights, making them basically almost always capable of fitting more complex patterns than appearing in the data. Therefore, at some point during training, ANNs start to heavily fit random noise, introducing a high variance error. This effect is called *overfitting*. [8, p. 107-113]

- Due to neural networks being able to have an arbitrarily amount of weights and the computational effort of the training increasing with each additional weight

specified in an network architecture, the runtime (and power cost) for finishing training can be extremely high (several days are not rare, even though there is no real upper bound). Additionally bigger datasets (10.000 data points) are also required compared to models of the more classic statistics domain. [6]

- Several layers on top of each other make interpretability of results extremely difficult and are an art of its own to master. [12]

Neural networks are very important for reinforcement learning as they allow to approximate unknown functions, like the Q function. In the *rlR* package they are put into heavy use - for example *AgentDQN* uses a fully connected network with two layers, while *AgentActorCritic* uses even two networks with several convolutional layers. [15]

2.3.3 Training

Like already mentioned the entries of every weight matrix and bias vector in each layer are initialized randomly, meaning that - assuming we want predictions to be as accurate as possible - these weights have to be changed in some way to not only give random predictions. For this we define a criterion function E (called loss function) being able to quantify the difference between the calculated prediction $\hat{y}$ of the network and the real target value y of each data point. To furthermore specify the training process, we only take a small sample of the whole dataset per training iteration (called *minibatch*, often in the size of 50 to 64 datapoints) - as it is empirically verified [8, p. 272] - this will not reduce the training quality, but greatly increase processing speed. After forward propagating the minibatch and recieving the predictions, calculating $\hat{E}$ on the minibatch data gives a good idea of how wrong these predictions are. It is now possible through calculating the derivates of E based on each weight to change the weights in a way to make these predictions less wrong or rather more accurate. Calculating the derivates of $\hat{E}$ is the most computationally expensive part of the whole training but thanks to the recursive nature of artifical neural networks, it is also possible to recursively calculate these by the chain rule. This process is called *backpropagation* and basically uses the derivates of each layer twice: One time for adapting the weights of the current layer (starting at the last layer) and the second time to give an interim result for calculating the derivate of the layer in front and therefore reducing the computational cost for the latter [8, p. 197-217]. After the derivate for a single weight is now given, its value is changed based on a procedure called *gradient descent* (or *stochastic gradient descent* in this very specific case, since we never use the whole dataset but rather small samples - the minibatches) [8, p. 286]. In this procedure the weight recieves a small change to its value in the direction of the gradient given by the derivative. A hyperparameter specifying the extent of the change of the value is given by ϵ.

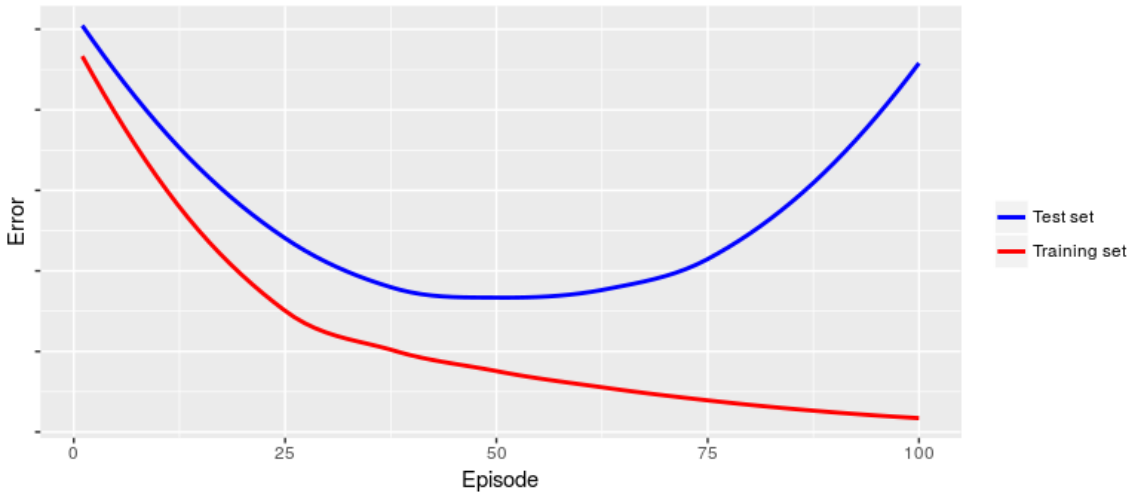

Figure 9: Generic example of overfitting appearing around episode 60 and higher during training phase

Formulating this process in pseudo code (for each weight w in the network) [10, p. 80f]:

$$\triangle w = -\epsilon \frac{\partial E}{\partial w}$$
$$w_{new} = w_{old} + \triangle w$$

The minibatch sampling, forward propagation, error calculation, backpropagation and weight adaption by stochastic gradient descent happens in every training episode in this order. The episodes happen iteratively and can theoretically go on forever even if only marginal changes in the weights happen. But it is important to keep overfitting in mind, so a typical stop criterion is once the loss function of a seperate test data set not being used during the training starts increasing [8, p. 107/112]. At first, the loss functions of the training and test data set looks roughly similar, but at some point the curve coresponding the training data keeps falling, while the other curve stales and slowly starts increasing again. At this point, it is save to assume further training will not improve the predictive capabilities of the network on new data points and therefore shall be stopped. In figure 9 one can see a training procedure with overfitting starting at around episode 60.

2.4 Deep Q learning

Now, we can use an artificial neural network to approximate the Q function stored in a tabular before. This makes it possible to apply Q learning on tasks with quasi infinite and high dimensional states. This combination of deep learning and Q learning also leads to its name - deep Q learning. Furthermore, because we need minibatches to calculate the derivates for the neural network training procedure, we have to use several state-action-reward-state combinations at once. But using consecutive

instances of these by always taking the last experienced ones introduces heavy correlation between the data points of a minibatch and may possibly skew the training and the end result. Therefore, every state-action-reward-state encounter is stored in a table called *replay memory*, making it possible to draw random samples for the minibatch and minimalizing any possible correlation as a consequence. Implementing these random samples with a replay memory into an algorithm is called *experience replay* and presented in the following pseudo code with the deep Q learning algorithm using the ϵ-greedy policy [11]:

Algorithm 1 Deep Q learning with experience replay

Initialize replay memory $\mathcal{D}$
Initialize neural network (representing action-value function Q) with random weights θ
for $episode = 1, \ldots, M$ **do**
    Initialise start state s_1
    for $t = 1, \ldots, T$ **do**
        With probability ϵ select a random action a_t
        Otherwise select $a_t = \max_a Q(s_t, a; \theta)$
        Execute action a_t and observe r_t and s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
        Sample random minibatch of transition (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{if } j \text{ is terminal} \\ r_j + \gamma \, max_{a'} Q(s_{j+1}, a'; \theta) & \text{otherwise} \end{cases}$
        Perform gradient descent step with $(y_j - Q(s_j, a_j; \theta))$ as loss function E
        Set $s_t = s_{t+1}$

This algorithm is also implemented in the R package *rlR* [15]. The neural network has two fully connected layers with a default amount of 64 hidden nodes. A demonstration of it is run in chapter 4.4.

3 Relational databases in R

3.1 Motivation

After an agent learned for a certain amount of episodes, and if the environment's states are defined as image data stored as three dimensional arrays in the computer memory, it can occure that the required storage is getting out of hand. In the special case of the OpenAI gym environment simulating Atari games, a single image frame has 210×160 coloured pixels with no alpha information (for transparency) - example in figure 10 depicted. This means we have $210 * 160 * 3 = 100800$ entries in such

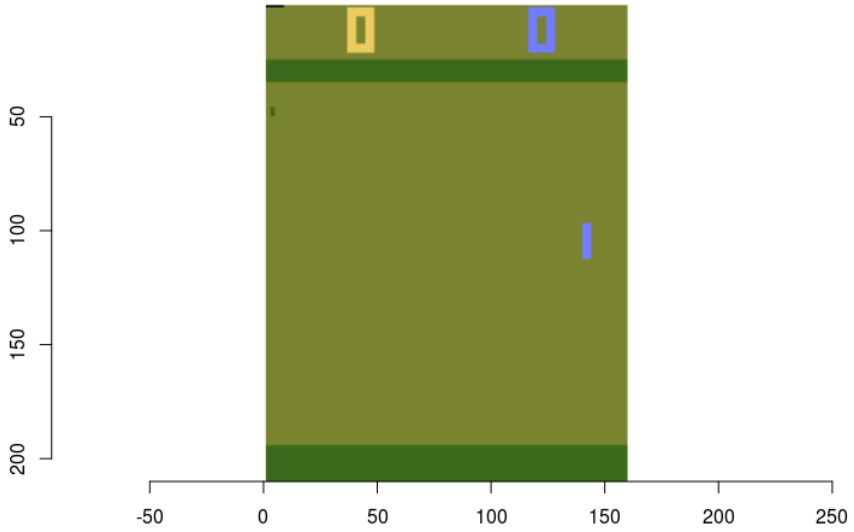

Figure 10: In R plotted image representation of the *Pong-v0* start state

an array holding integer values between 0 to 255. An integer in R always requires 4 Bytes of storage, creating a minimal required storage space for a single state of

$$100800 * 4B = 403200B$$

To confirm this with a simple example, let's construct a *Pong-v0* environment [2, OpenAI gym package] and check the storage size of a single state:

```
library(reticulate)
gym   = import("gym")
env   = gym$make("Pong-v0")
state = env$reset()  # generate the starting state
object.size(state)
#> 403408 bytes
```

As we can see a single state does indeed require approximately 400 KB of storage. After an agent finished one episode of training, the amount of states visited and stored in the replay memory may easily exceed the 1000 mark. Even if it's designed smart and doesn't store every old and new state as a pair, but rather use state IDs for not saving every state twice, the required storage hits

$$400KB * 1000 = 400MB$$

for sure.
Since several episodes are required to be finished to make an agent learn an optimal solution for the environment, it's highly realistic to assume several Gigabytes of storage size are required only for the replay memory. This leads to the danger of

the whole session to crash if the RAM limit of the used PC is exceeded. A already really good PC has 16 GB of RAM, making it possible to store roughly 40 episodes of the above example at best before crashing. Empircal testing on a single machine with this amount of RAM showed that the amount of 40 episodes was indeed never reached once - the R session always aborted before.
Due to Atari games being very old, the above storage requirements don't even hold true for most of todays image generation processes. Alhough there's definitely a limit of image pixel amount a state will have in today's standards and reducing the pixels by resizing the image (and thus reducing the image quality) is always a possibility, creating a solid storage strategy for almost arbitrarily large states is key to successfully implement a replay memory for any Reinforcement Learning algorithm requiring such one.
Additionally it may be desired to reliably store a replay memory independently of sessions and to manage its entries in an accessable way. Therefore a solution for these problems is presented in the following. It consists of shifting the storage from the working memory to the SSD by using the relational database management system SQLite. For this a theoretical part about relational databases and SQLite will be presented before going through the details of the implementations and the read/write/storage performance benchmarks in this chapter.

3.2 Basic theory of relational databases and SQLite

The underlying theory for relational databases is the relational algebra.
In the relation algebra relations are defined as a subset of n-tuples resulting from n domains [1, p. 73]:

$$R \subseteq D_1 \times D_2 \times \cdots \times D_n$$

For example, if you have two relations (by definition every domain is also a relation and every relation is also a set) A and B with each containing 1-tuples, then the cartesian product of these is a 2-tuple:

$$A \times B = \{(a, b) | \, a \in A, b \in B\}$$

Furthermore the set operations union, intersection or difference also work the same on relations as long as these relations have the same tuple size. Thus, $(A \times B) \cup A$ is not valid for relations, if the result has to be a relation again.
Since dealing with complete relations is unpractical, two more relation specific operations are required to narrow down the results we recieve by looking at a relation:

- Selection [1, p. 88]:

$$\sigma_{condition}(R) := \{r | \, condition(r) \wedge r \in R\}$$

The σ operator allows to make a selection on the tuples in a relation. In general the condition can consist of a (valid) mixture of arbitrarily complexity of the following: Domains of R, constants, boolean operators, arithmetic equation operators.

- Projection [1, p. 89]:

  $\Pi_{D_{k_1},\ldots,D_{k_l}}(R) \coloneqq \{(d_{k_1},\ldots,d_{k_l}) | \, (d_1,\ldots,d_{k_1},\ldots,d_{k_l},\ldots,d_n) \in R\}$, with $k_1 < \cdots < k_l$ and $l \leq$

  The Π operator allows us to make a selection of the domains a relation consists of.

For our usecase these theoretical constructs are already enough to successfully understand the commands in a relational/SQL database [1, p. 118]:

- tables in a database $\iff$ relations
- columns of a table $\iff$ domains of a relation
- SELECT statement $\iff$ *Projection*-Operator
- WHERE statement $\iff$ *Selection*-Operator
- FROM statement $\iff$ argument of the above operators

For the database implementation in this thesis the choice to use SQLite as a database management system was made for several reasons. First of all it's consistent with the ACID-Principle (an acronyme for these traits) [4]:

- **Atomicity**:
  Every transaction consisting of multiple statements will only have **any** effect if and only if every single statement executes successfully. So, if one statement fails, the whole transaction will fail, leaving the database unchanged.
- **Consistency**:
  Every transaction can only move the database from one valid state to the other - changes in data are either consistent with all defined rules or not made at all. Note that single statements of a transaction may lead to invalid intermediate steps, but due to the Atomicity principle these will be reversed if the transaction fails.
- **Isolation**:
  Transactions executed in parallel need to have the same results as if executed sequentially - otherwise one transaction will be blocked until the other finishes.
- **Durability**:
  Once a transaction finishes, any changes done to the database are durable - even in the case of a crash or power outage.

These principles guarantee a database free of corrupted data. Furthermore, SQLite is open source and follows a minimalistic principle, resulting in a very easy installation process and low storage requirements. Thus, making SQLite perfect as a tool comming with an R package, even though (or just because) SQLite is only meant for single user usage. [5]
To demonstrate SQLite with R the package *RSQLite* will be used in the following with some basic examples. First of all, after loading the library in R, an object containing the connection to a database (here: *my_database*) has to be aquired:

```
library(RSQLite)
con = dbConnect(SQLite(), dbname = "my_database")
```

The *con* object will then be used in combination with any statement doing a transaction in the database of this connection. Here, we start by creating a new table scheme called *my_table* [1, p. 115]:

```
create_table = "
  CREATE TABLE my_table (
    matriculation_id INTEGER PRIMARY KEY,
    name             TEXT    NOT NULL,
    age              INTEGER NOT NULL CHECK(age > 0),
    registration     DATE    DEFAULT(CURRENT_DATE)
  )
"
dbExecute(con, create_table)
#> 0
```

The listings in the brackets are the column names of the table with their data type and further constraints of arbitrarily amount. *PRIMARY KEY* makes this column only hold unique values, allowing the user to uniquely identify rows by these values. Additionally, *CHECK* only allows entries meeting the condition of its argument, while *DEFAULT* sets a default value if no value is specified for new entries.
After creating the hull of our table, let's fill it with some life with the *INSERT*-statement [1, p. 116]:

```
insert_values = "
  INSERT INTO my_table (name, age)
                VALUES ('Max', 18),
                       ('Berta', 19)
"
dbExecute(con, insert_values)
#> 2
```

The brackets behind the table name and the *VALUES* keyword require the same amount of entries, as this statement maps values to columns, while filling unmentioned columns with *NULL* or the predefined default value. Note, by not declaring

name or *age* one would trigger an error in this case due to the *NOT NULL* constraint.
Now, to get the data from our table, we write a query by applying the above defined relational logic [1, p. 116]:

```
query = "
  SELECT *
  FROM my_table
  WHERE age = 19
"
dbGetQuery(con, query)
#>   matriculation_id  name age registration
#> 1                2 Berta  19   2018-06-22
```

This query just selected all columns (by using the special character *́instead of column names) of the table *my_table* with the entry in the *age* column having a value equal to 19.
We can also use more convenient wrapper functions of the *RSQLite* package to write data to a table without defining its scheme, even though not defining a table scheme is absolutely not recommended since no constraints (like *PRIMARY KEYS*) can be specified, making undesired entries possible. An example of use of such a wrapper function:

```
dbWriteTable(con, "cars", mtcars)
cars = dbGetQuery(con, "SELECT * FROM cars")
all(cars == mtcars)
#> TRUE
```

The last line tells us, by writing to the database table and querying the entries again, nothing changed - like it is supposed to be.
To delete unwanted tables simply use the *DROP TABLE* statement:

```
dbExecute(con, "DROP TABLE my_table")
#> 0
dbExecute(con, "DROP TABLE cars")
#> 0
```

No confirmation or delay will happen by executing the above lines, so use with care.

4 Database and visualization method implementations for reinforcement learning in R

In the following sections the implementations of a database into the R package *rlR* and the plots of an app for visualization (also implemented into *rlR*) will be presented.

4.1 Database design

The states of the *Pong-v0* environment are represented as an three dimensional array with 100800 entries. This is extremely inefficient for storing image data. For this reason a method for lossless data compression was chosen by transforming the image data to the PNG format with the help of the R package *PNG* [13]. The data is then represented as a string consisting of appended hexadecimal value pairs.

```
library(png)

(state_png = writePNG(state / 255L) %>% paste(collapse = "")
   )
#> [1] "89504e470d0a1a0a0000000d[...]"
nchar(state_png) # length of the string
#> [1] 1408
object.size(state_png)
#> 1504 bytes
```

As we can see the storage requirements for a single state is dramatically reduced by a factor of approximately 270 compared to storing the state in its ńaturalárray format.
To confirm there is no loss of information by running the state through all of these steps the following code was executed. The last line of code basically gives out that the values of the state did indeed not change at all.

```
parser = function(x)
  paste0(x[c(TRUE, FALSE)], x[c(FALSE, TRUE)]) %>%
  as.hexmode %>%   # necessary for correct as.raw
  as.raw %>%       # make it readable as PNG
  readPNG * 255L

con = dbConnect(SQLite(), dbname = "replay_memory")
dbWriteTable(con, "test_png", data.frame(state = state_png))
result = dbGetQuery(con, "SELECT * FROM test_png")[[1]] %>%
  strsplit("") %>%
  .[[1]] %>%
  parser

all(result == state)
#> [1] TRUE
```

4.2 Database benchmarks

To test the database performance a *Pong-v0* environment of the gym package was run for n amount of steps (with n being set to 100, 1000 and 10000). The steps were made by a random agent, but the rest was kept as closely as possible to a real application. During every step the information (iteration, current state, action, reward, next state) was stored as a single write transaction into a predefined table in the replay memory database. The required time to finish these n steps was noted. The computer, the benchmarks have been performed on, is equipped with an SSD, therefore making the ratios to in-memory versions totally overoptimistic for using an HDD.

First of all it's important to define the table scheme manually, so the *state_id* we are using for identifying every entry is unique (due to specifying it as *PRIMARY KEY*):

```
dbExecute( con, "
  CREATE TABLE IF NOT EXISTS pong_png (
    state_id    INTEGER PRIMARY KEY,
    state       TEXT,
    next_state  TEXT,
    reward      NUMERIC,
    action      INTEGER
  )"
)
```

If we search in a column with non-unique entries we have to check every entry for equality (also known as exhaustive search), thus resulting in an average time complexity of $O(n)$ for a single search on a column with n entries.
But if we search in a column with only unique entries, SQLite performs a binary search - for every entry checked half of the unchecked entries are discarded, resulting in an average time complexity of $O(logn)$ for the whole search of a single entry. [7, p. 135]
So by defining the *state_id* as *PRIMARY KEY*, the time complexity of searching for specific entries by their *state_id*'s is reduced from $O(kn)$ to $O(k \log n)$ for k amount of searches on a table with n entries. [7]

The R code for a single write into the table *pong_png* of the connection *con* (executed n times):

```
dbWriteTable( con, "pong_png",
  data.frame(
    state_id = iteration,
    state =
                  writePNG(old_state / 255L) %>% paste(
                    collapse = ""),
    next_state =
                  writePNG(observation / 255L) %>% paste(
                    collapse = ""),
    reward   = reward,
    action   = action
  ), append = TRUE
)
```

After n steps were performed, a single read over all n entries of the table was performed and also measured:

```
entries = dbGetQuery( con, "SELECT * FROM 'pong_png'" )
```

To make the measurement of the transaction above even in confidence with the other transactions the mean was taken of n repetitions. Additionally all entries were called separately by indexing of the iteration leading to n single reads being measured together:

```
for (index in 1:n)
  entry = dbGetQuery(
    con,
    paste("SELECT * FROM 'pong_png' WHERE state_id =", index
       )
  )
```

These three measurements were repeated ten times and the average results in seconds are represented in the following table for $n = 100, 1000, 10000$:

n	Write	All read	Repeated single read
100	1.3402 s	0.0009 s	0.0302 s
1000	13.5244 s	0.0078 s	0.3257 s
10000	136.9608 s	0.0723 s	3.5405 s

Besides that a SQLite specific analysis program [3, sqlite3_analyzer] was used to further check the storage requirements after 10000 entries were written. The conclusion was that approximately 41 MB of overall storage was consumed with a single entry requiring 4.1 KByte.

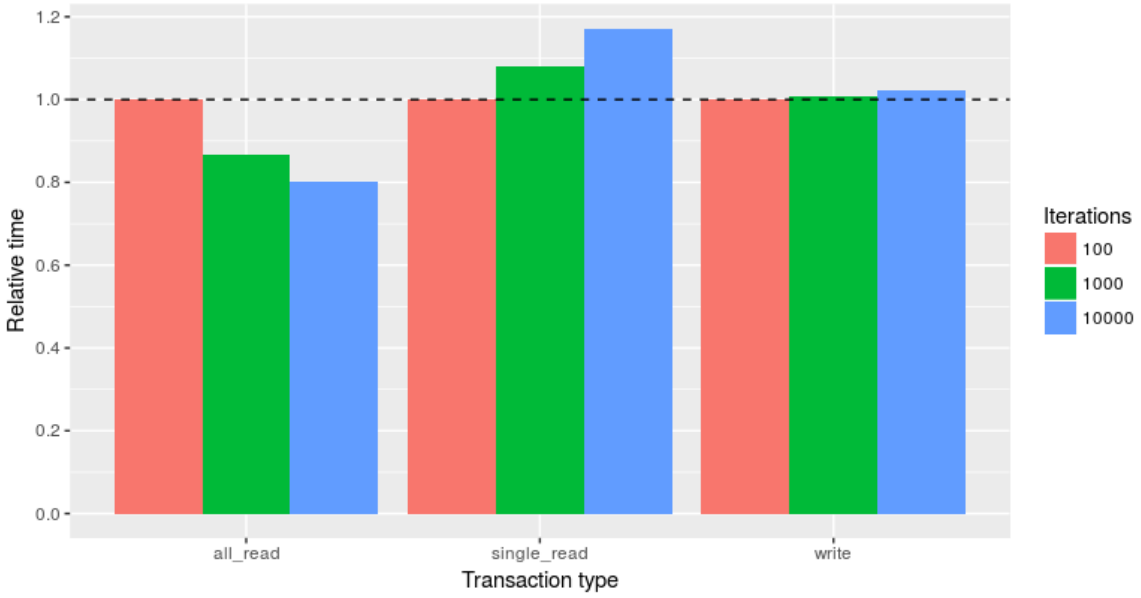

Figure 11: Measured relative time of a single transaction (with 100 iterations of each type as baseline)

Besides that, a head to head benchmark between the implementations in *rlR* with and without database was done. This way it's possible to estimate the impact of writing and reading SQLite databases on the overall runtime compared to the time an agent requires for fitting its model. Note the *Pong-v0* runs finishing a lot faster due to the R session crashing thanks to running out of memory for higher learning episodes.

The following tabular holds benchmarks (measured in seconds) of different setups - comparing the usage of a database with without one in each. The *CartPole-v0* environments have run for exactly 1000 episodes, while the *Pong-v0* have run for only 10 episodes. This is because sessions of *Pong-v0* crashed on the test computer (with 16 GB RAM) due to running out of memory for an higher episode amount. *AgentPG* is another reinforcement learning algorithm called monte-carlo **P**olicy-**G**radient implemented in the *rlR* package, that is only adapting its parameters after a full episode finished, therefore performing way less sampling than the **D**eep **Q** **N**etwork/learning agent.

Environment	Agent	No Database	Database
CartPole-v0	AgentPG	421 s	492 s
	AgentDQN	2150 s	4116 s
Pong-v0	AgentPG	227 s	326 s
	AgentDQN	4018 s	7173 s

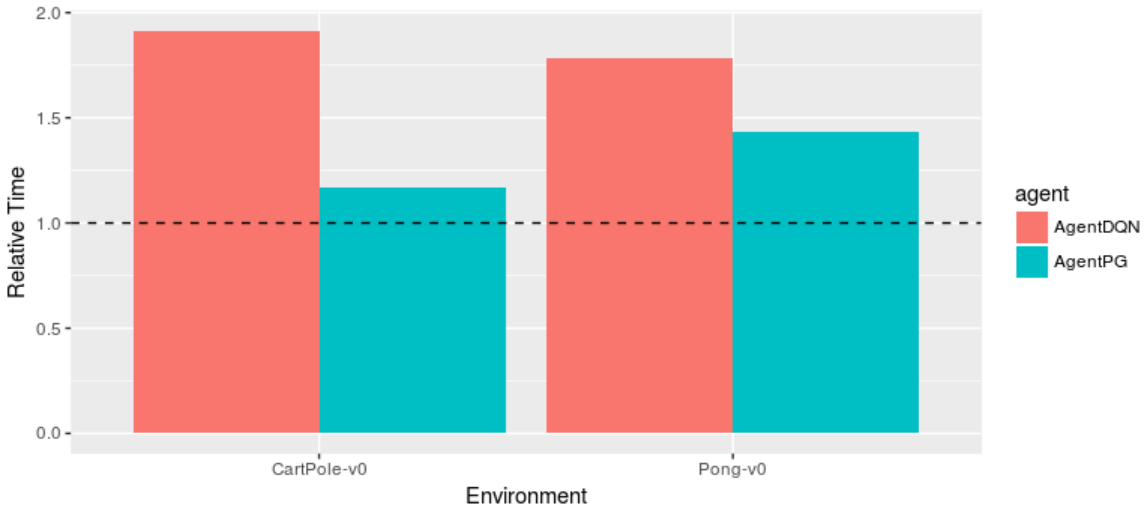

Figure 12: Measured time of the database versions relative to each version without database

On average approximately 89 entries were written per episode of *CartPole-v0*, while for the *Pong-v0* environment the amount was 1050. This is important to put into perspective because single indexing of the tables was heavily used, thus the runtime for a single search increases the more entries (n) there are in the table (due to $O(log\, n)$ search complexity).

4.3 Evaluation

Even though the chosen database is highly scalable and performs well for very high entry sizes, based on the benchmarks, we can conclude the runtime is in general suffering from the database implementation. But it is also depending heavily on the algorithm and the environment being used to what extend this time punishment appears. It seems in cases of image processing environments (like *Pong-v0*) the runtime disadvantage is not as bad as in the case of low dimensional states (like *CartPole-v0*). Furthermore, the agent has an even bigger influence. The cause for this is simply how often the agent performs sampling of the replay memory in form of index searching - *AgentDQN* namely performs a sampling after every step during an episode, while *AgentPG* only samples after an episode. This means the latter does fewer transactions than the former, which at no surprise leads to less additional time required for using the database instead of the computer's working memory. As a conclusion one can say that it is very case dependent if the use of a database makes sense. The additional time required to run one is not neglectable for really long training sessions - requiring for example four days instead of two to finish can be a big no-go for some situations. But, if the replay memory is outgrowing the capabilities of the PC's memory, then there will be no way around using one. So it

may make sense to check the size of a single state beforehand and roughly calculate an estimate if the algorithm finishes converging before the working memory is full.

4.4 Visualization

Another important implementation done for this thesis is a way to interactively visualize the training progress of the action value function predictions and to have a view of how the action value function is shaping over the whole state space. Additionally, the option to view the trained weights of the used neural network is also given. This all is provided by designing a web application with help of the R package *shiny* one can start by running a normal learn session of the *rlR* package. Due to the plots used being low dimensional, the app is not designed (and indeed will not start) for image data.

In the following a basic example for demonstrating the visualization capabilities of the app was created. First of all, the *rlR* package has to be loaded and an agent with an environment set up. In this case, the choice was fallen on the *CartPole-v0* and the *AgentDQN* as these were already used in advance:

```
library(rlR)

env  = makeGymEnv("CartPole-v0")
conf = getDefaultConf("AgentDQN")
conf$set(agent.store.model = TRUE)
agent = makeAgent("AgentDQN", env, conf)
perf = agentDB$learn(300)
```

As one can see in figure 13, the agent successfully converged during these 300 episodes, so it makes sense to now have a look at the insights of the training process the app can provide.

For this, the visualization app can be started by running the following method *startApp* on the performance object *perf* returned by the *learn* method in the code snippet before:

```
perf$startApp()
```

The app contains 4 tabs with different interactive plots (created with the package *plotly*): a 2D and a 3D plot each for weights of the neural network and for the action value function. In the following screenshots of the interactive 3D plots are shown, although the real graphics in the app are superior at delivering insights.
In figure 14 and 15, the values of the weight matrix entries of the first layer are visualized by their respective row and column index in the matrix. For example, this means all dots at $col\_index = 4$ are the weights connecting input dimension 4 with each hidden node. Similar for the dots on $row\_index = 1$ presenting the weights (or coefficients in another context) of the linear combination of the input vector for

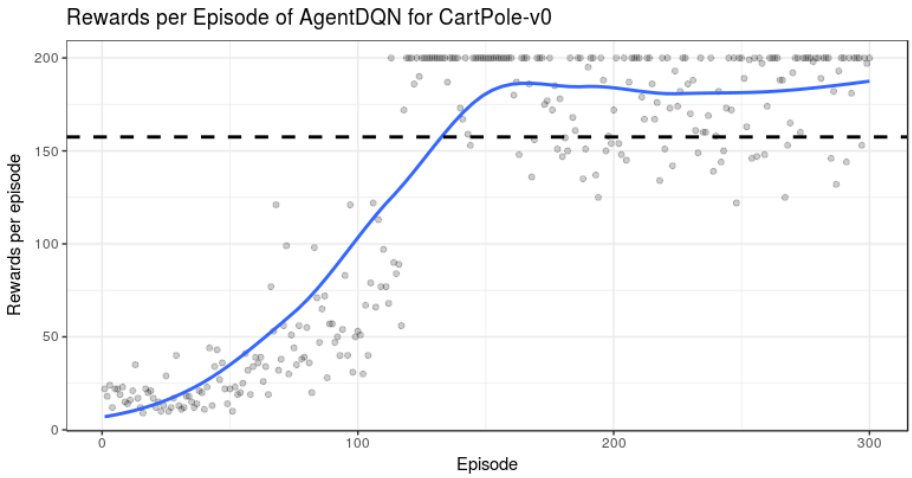

Figure 13: Rewards per episode over time for 300 consecutive learning episodes of an DQN agent in the CartPole-v0 environment.

hidden node 1. Same goes for figure 16 and 17 showing the respective weight values of the second layer. As a quick side note to avoid confusion: The colours in the plot are redundant in information as they are simply set by the value of the Z-axis - the choice was still made to keep them like they are, because turning around the plot interactively in the app may lead to a confusion in perspective. Of course this issue does not appear in static screenshots like presented here.

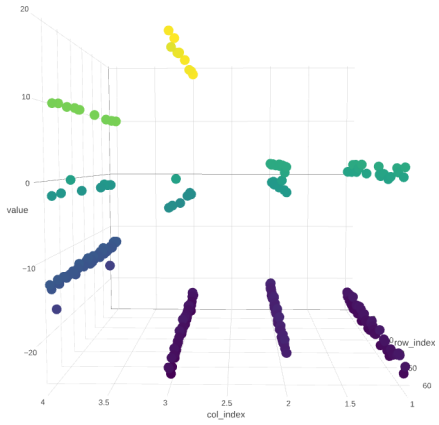

Figure 14: Weight values of the first layer

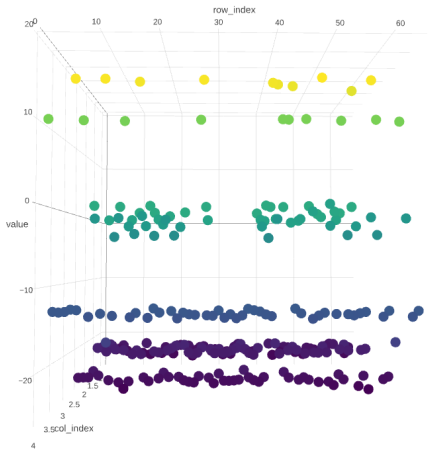

Figure 15: Weight values of the first layer - different perspective

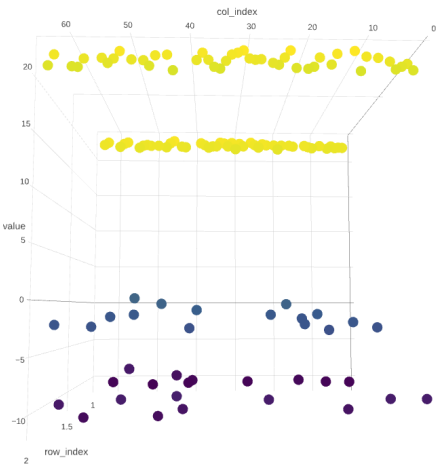

Figure 16: Weight values of the second layer

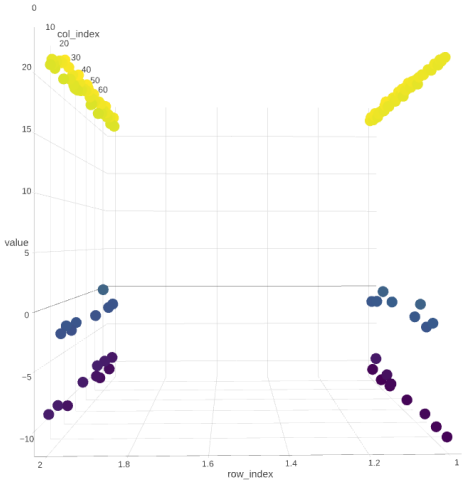

Figure 17: Weight values of the second layer - different perspective

Furthermore, the app also provides the possibility to plot the prediction surface of the action value function dimensions in respective to the state space. For this, the UI of the app gives several options to see the by the user desired information. Extensive screenshots of this are shown in the appendix. For example, one can set the following options amongst other things:

- the Z-axis to a desired action value dimension
- the X- and Y-axis to desired state dimensions
- the prediction plane after a specific episode
- the values of state dimensions not occuring in any plot axis

Additionally, the graphic holds the observed values to give an idea of how well the predictions represent the existing data points. Because after several hundreds of episodes the amount of observations is way too high for clear plotting, the choice was made to only pick the data points of the last batch sample (in this case with size 64).
Figures 18, 19, 20 and 21 show the development of the shape of the prediction plane based on different learning episodes. In these plots the Z-Axis was chosen to be the first action value dimension (in this environment representing the léftáction) and the X- and Y-Axis to represent the first and second state dimensions - the other state dimensions are set to the value **0.0**. As it is depicted, the surface of the prediction plan becomes more and more extreme in its slope and manages to approximate the data points in the last batch quite well.

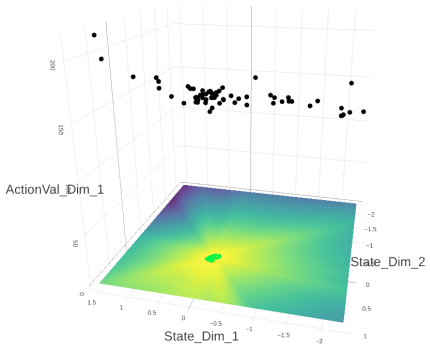

Figure 18: After one episode: Prediction plain of the action values for the action *left*

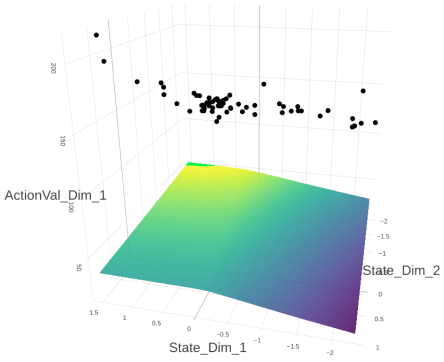

Figure 19: After 100 episodes: Prediction plain of the action values for the action *left*

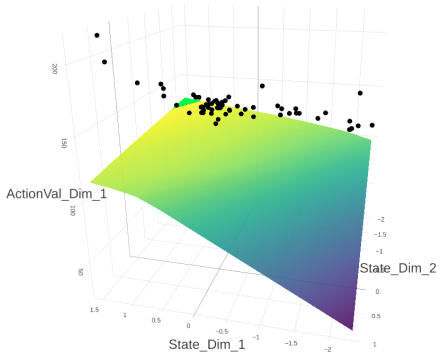

Figure 20: After 200 episodes: Prediction plain of the action values for the action *left*

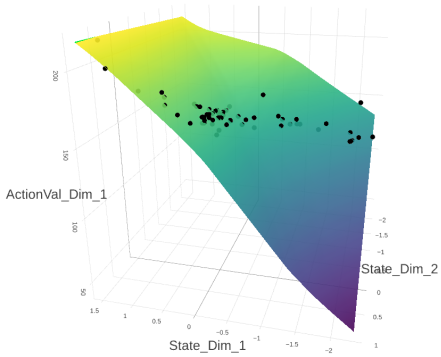

Figure 21: After 300 episodes: Prediction plain of the action values for the action *left*

5 Summary

In this thesis an overview of the basic theories of reinforcement learning, artificial neural networks and relational databases was given with the addition of a basic tabular Q learning implementation. Furthermore in the context of the *rlR* package, a database scheme for the replay memory was designed and benchmarked, and a visualization app for training insights has been created. The database solution is primarily a valid choice for image data to handle the high amount of storage requirements, but it may not be the best choice for low dimensional data in combination with learning agents doing lots of sampling, as the benchmarks showed that the additional runtime required is considerably high. Besides that, the visualization app allows to look into the learning process of an agent in detail even in hindsight, allowing to understand better why and how convergence happend - or didn't happen. This may be especially important, because it is often difficult to understand the behaviour of algorithms in the reinforcement learning domain, where the data the agent produces by his behaviour is also as important as the machine learning model that tries to fit these data points.

A very interesting and expectedly highly useful further addition (that was out of the scope of this thesis) to the app would be to provide an interactive plot showing the decisions in the complete state space the agent would make based on its policy with the option to select an arbitrarily episode. This would make it even easier to reason about the agents behaviour, as currently only one action value dimension can be plotted, making it hard to compare different actions in their value and thereby what action is preferably.

References

[1] Andr Eickler Alfons Kemper. *Datenbanksysteme: Eine Einführung*. De Gruyter Studium, 10. edition, 2015.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] SQLite development team. Sqlite 3 analyzer. `https://www.sqlite.org/download.html`. abgerufen am 20.07.2018.

[4] SQLite development team. Sqlite is transactional. `https://www.sqlite.org/transactional.html`. abgerufen am 20.07.2018.

[5] SQLite development team. Sqlite overview. `https://www.tutorialspoint.com/sqlite/sqlite_overview.html`. abgerufen am 20.07.2018.

[6] Chris Edwards. Growing pains for deep learning. *Commun. ACM*, 58(7):14–16, June 2015.

[7] Stefan Güting, Ralf Hartmut abd Dieker. *Datenstrukturen und Algorithmen*. Teubner, 3. edition, 2013.

[8] Aaron Courville Ian Goodfellow, Yoshua Bengio. *Deep Learning*. MIT Press, Cambridge, MA, 2016.

[9] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[10] David Kriesel. *Ein kleiner Überblick über Neuronale Netze*. Unknown, 2007.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[12] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *CoRR*, abs/1706.07979, 2017.

[13] Greg Roelofs. Png intro. `http://www.libpng.org/pub/png/pngintro.html`. abgerufen am 20.07.2018.

[14] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354 EP –, Oct 2017. Article.

[15] Xudong Sun. R package rlr. `https://github.com/smilesun/rlR`. abgerufen am 20.07.2018.

[16] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2. edition, 2017.

[17] Dat Tran. Why the ai hype train is already off the rails and why im over ai already. `https://builttoadapt.io/why-the-ai-hype-train-is-already-off-the-rails-and-why-im-over-ai-already-e7314e` abgerufen am 20.07.2018.

Appendices

Figure 22: User interface 1

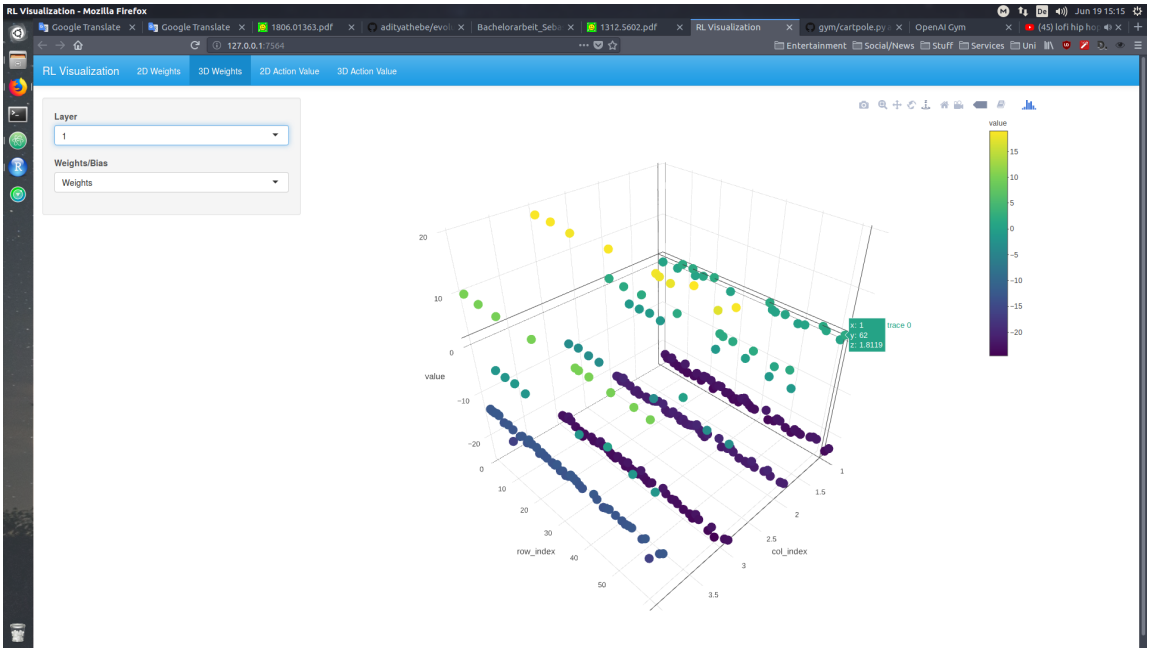

Figure 23: User interface 2

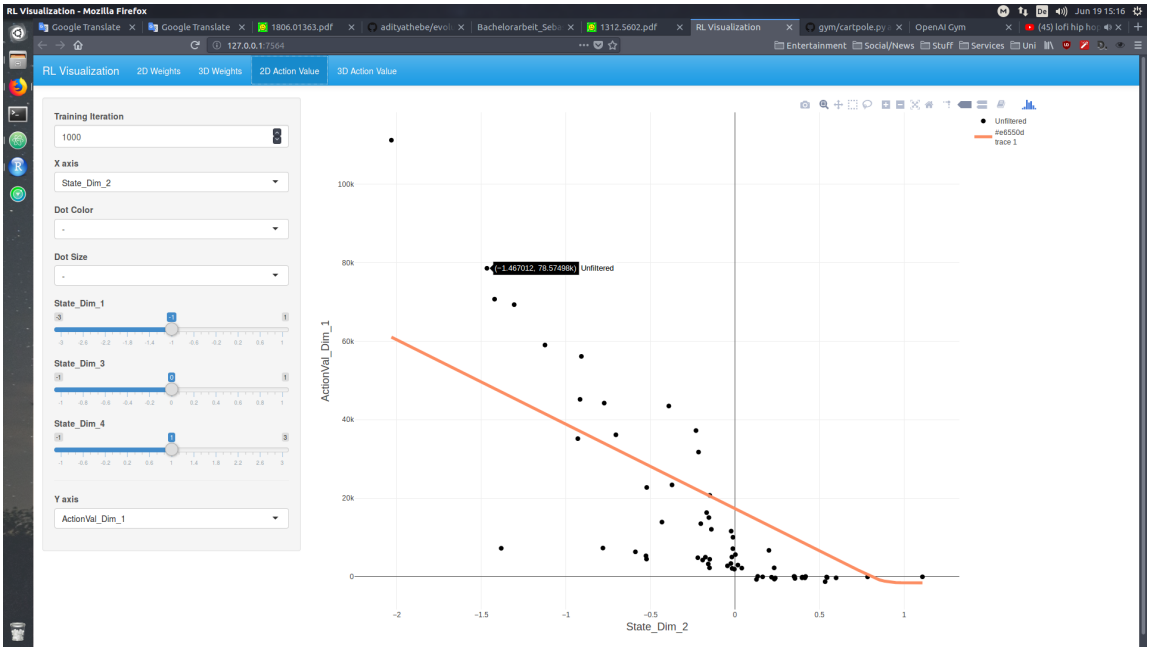

Figure 24: User interface 3

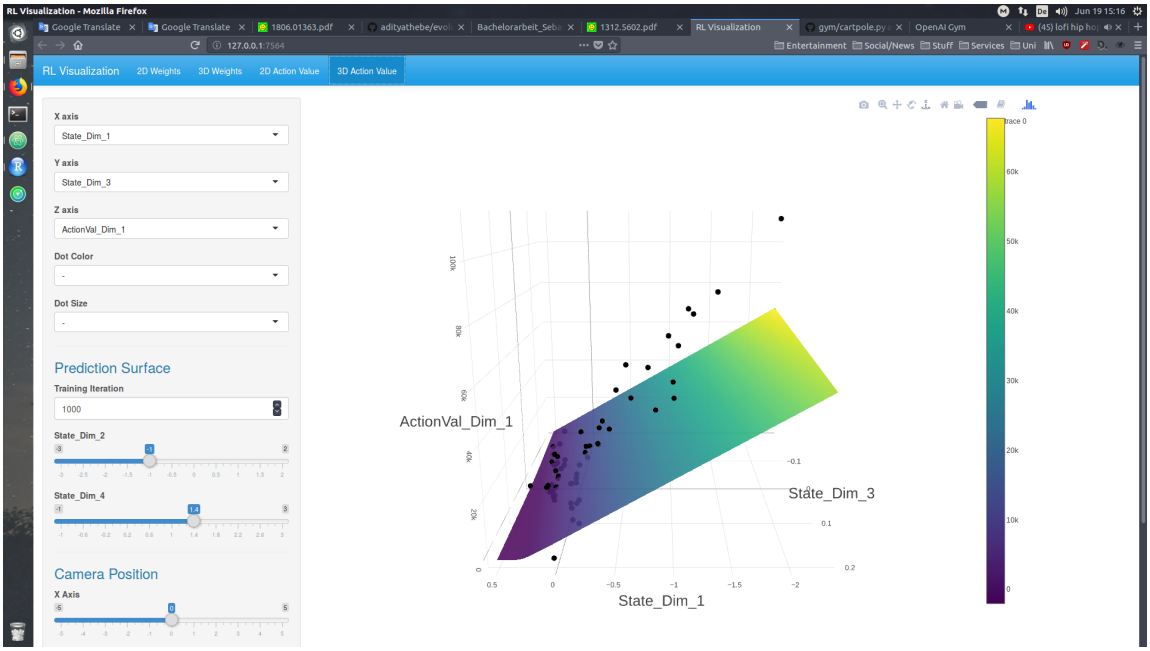

Figure 25: User interface 4

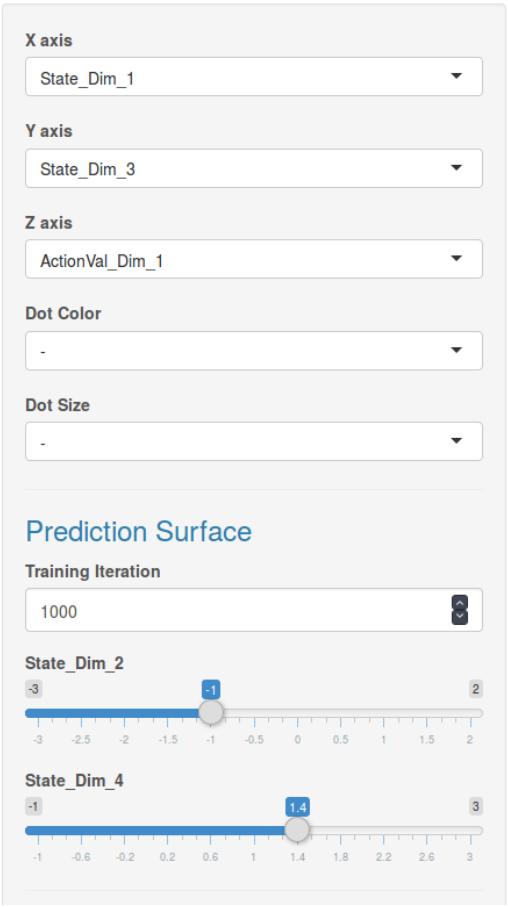

Figure 26: User interface 5

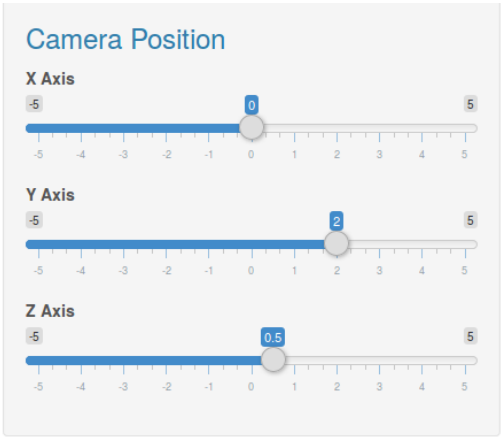

Figure 27: User interface 6

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Die Arbeit wurde weder einer anderen Prüfungsbehörde vorgelegt noch veröffentlicht.

München, den 21.07.2018

.....
Sebastian Gruber